TÉCNICO LISBOA

Instituto Superior Técnico

Mestrado em Engenharia Electrotécnica e de Computadores

Parallel and Distributed Computing

Project Report - OpenMP

Gonçalo Bernardino Frazão

Gustavo Rodrigues Vítor

João Paiva de Castelo-Branco

Prof. José Monteiro

2023/2024

1 - Introduction

This report covers the second part of the Parallel and Distributed Computing project for the academic year 2023/2024. The project involves creating a 3D version of Conway's Game of Life using different programming approaches: serial implementation, using OpenMP, and using MPI. This document focuses on the OpenMP approach. It starts with a quick review of the serial implementation before diving into a comprehensive analysis on the OpenMP-based program.

2 - Serial implementation

The serial implementation was reasonably straightforward. The program is executed with the four requested arguments on the command line and begins by dynamically allocating a 3D grid with dimensions N^3, representing a cube. Following the requirements outlined in the problem statement, the simulation is executed and its duration timed. The results are displayed on `stdout` in the format:

```
<cell species> <maximum number> <generation where this max occurred>
```

Finally, the execution time is printed to `stderr` as a performance metric. All the available tests on the course web page were passed with highly satisfactory timings and correct outputs. These results will be shown later in the report and compared with the results from the OpenMP implementation.

Simulation Algorithm

The simulation itself employs a brute-force approach. For each generation, the states of cells are determined by examining the 26 neighbors of each cell in the 3D grid. Two vectors store important data throughout the simulation: one for the maximum number of cells of each species reached in any generation, and the other for the corresponding generation where this occurred.

The simulation initiates by counting the number of cells for every species in generation zero. The process then iterates through to generation N, performing the following steps:

- A vector of size `NSPECIES` + 1 is declared (with index 0 used for dead cells) to count the cells for the next generation.
- Iterate through every cell of the current generation using three nested loops:
  - Compute the next state of the current cell based on its 26 neighbor cells using the `next_state` routine.
  - Increment the previously mentioned vector by one at the index corresponding to the cell's species. This approach avoids the need for an additional iteration through the entire grid (as required for generation 0) to count cells by species.
- Compare the number of cells of each species in the new generation with the maximum obtained in previous generations and update the global vectors if necessary.

An auxiliary 3D grid is used to store the states of the next generation's cells, meaning two grids of size N^3 are kept in memory and switched between to represent the current and next generation's states.

This implementation was continuously optimized to achieve significant computational efficiency. For further details, please refer to the provided code and its accompanying comments.

3 - OpenMP Implementation

3.1 Approach for Parallelization

The parallelization strategy employs OpenMP directives to enhance the efficiency of the simulation. While the original algorithm from the serial implementation is retained, parallel constructs are introduced to improve processing speed. This approach maintains a consistent basis for comparison with the serial version and allows for accurate speedup calculations.

Given the nature of the problem, the computation of cell states for each generation remains inherently sequential, as each state relies solely on conditions from the preceding generation. However, the task of determining each cell's subsequent state transitions from a sequential to a parallel process through the application of OpenMP directives. This method effectively segments the grid, with individual portions assigned to separate threads. In this OpenMP-enhanced version, the directive `\#pragma omp parallel for reduction` optimizes the main simulation loop. Within this parallel region, each thread independently manages a segment of the grid, updating cell states without inter-thread dependencies and effectively eliminating the risk of data races.

Similarly, this parallel directive has been applied to the `count_cells` function. This enables the concurrent tallying of species populations across the initial grid configuration and streamlines the process. Consequently, this parallel counting significantly contributes to the efficient population of the `max_cells` array, thereby optimizing the initial data aggregation phase.

3.2 Decomposition and Load Balancing

The data decomposition strategy involves dividing the grid into smaller segments, with each one assigned to a different thread. Ideally, if there are `p` physical processors available, the grid should be divided into `p` equal parts for processing. This is done using the static schedule, which typically divides the cube into chunks of size $\frac{N^3}{p}$ when there are N^3 iterations to perform. This method reduces the overhead associated with thread creation, management, and termination, while ensuring an efficient distribution of computational work among threads.

However, based on the grid allocation routines provided by the teaching staff, the second and third dimensions of the cube (identified here as y and z) are stored contiguously in memory, as illustrated in Figure 1.

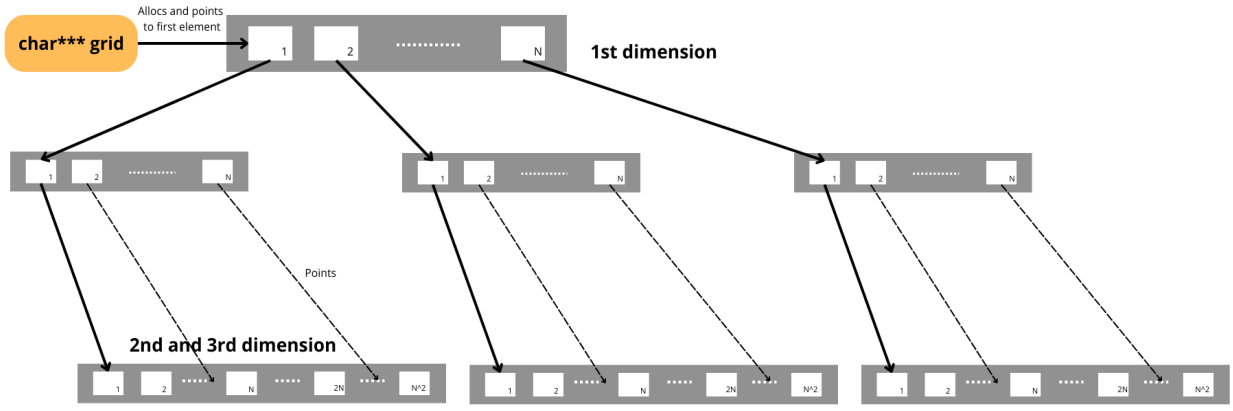


Figure 1: Schematic of 3D grid allocation in memory

Therefore, it's beneficial to divide the cube along the first dimension, x, because the y and z coordinates for any x are close together in memory. This arrangement improves the cache hit rate and speeds up access to memory data. Consequently, the cube is divided along its x dimension according to the number of threads set by the `OMP_NUM_THREADS` environment variable. If the division N/p is an integer, then the workload among threads will be evenly balanced. Nonetheless, with the same number of processors `p`, the larger the grid size `N`, the smaller the impact of any differences in load.

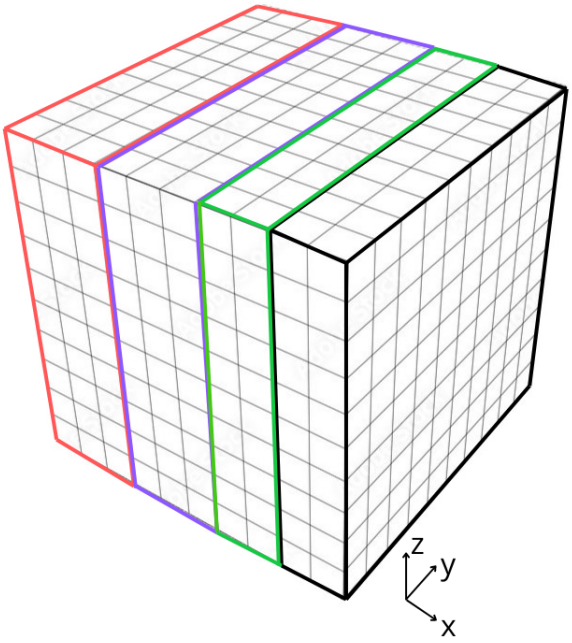


Figure 2: Schematic of 3D Grid Division for $N = 10$ and $p = 4$ threads using static schedule

3.3 Synchronization Concerns

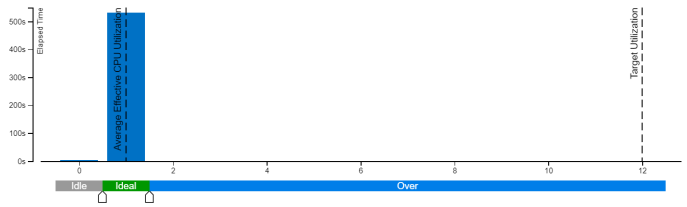
Without appropriate synchronization mechanisms, there is a risk that multiple threads could read and write to the same elements of the arrays concurrently. Such concurrent updates to shared data structures may lead to inconsistent states, resulting in incorrect or mismatched values across threads.

As previously discussed, a single generation is processed in parallel by multiple threads, while the transition from one generation to the next is handled sequentially. This ensures that a new generation begins processing only after the completion of the previous one. However, while computing the new or next generation, the new cell states depend solely on the neighbor states from the current generation, which have already been computed. This means there will not be any data races, even though different threads may read from the same cell simultaneously. This approach ensures proper synchronization without data inconsistencies.

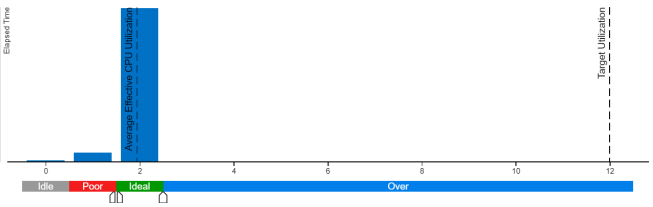
On the other hand, when processing a given generation, the counting of cell species occurs concurrently, potentially affecting the update of a global variable `cells` that stores species count. To address potential synchronization issues, OpenMP's reduction clause is used along with parallel constructs. This mechanism ensures that each thread operates on its own local copy of the `cells` array, accumulating its partial results independently. At the conclusion of the parallel region, these local results are combined accurately into the global `cells` array. This strategy prevents race conditions and maintains data integrity.

3.4 Performance Results and Analysis

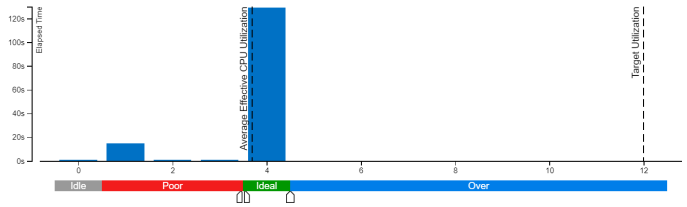
In this section of the report, we will be presenting the program's execution times, speedups, and effective CPU utilization for various thread counts. This analysis was performed using Intel's VTune Profiler on a machine equipped with 6 physical CPUs (with 2 threads per core):



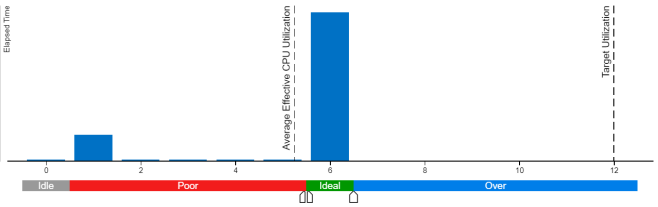
(a) Simultaneously utilized CPUs for 1 thread



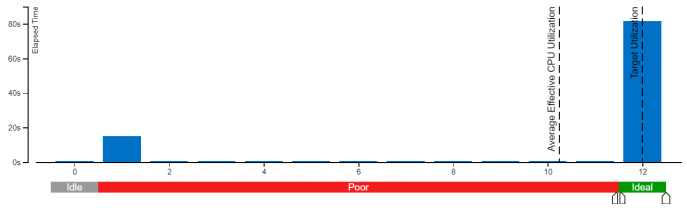
(b) Simultaneously utilized CPUs for 2 threads



(c) Simultaneously utilized CPUs for 4 threads



(d) Simultaneously utilized CPUs for 6 threads



(e) Simultaneously utilized CPUs for 12 threads

Figure 3: Effective CPU utilization for `./life3d-omp 3 1024 .4 100` with 1, 2, 4, 6 and 12 threads

Number of threads	**1**	**2**	**4**	**6**	**12**
Execution time (s)	516.8	258.8	129.3	86.6	82.9
Speedup	-	1.9969	3.9969	5.9677	6.2340
Speedup (%)	-	99.845	99.923	99.462	51.950

Table 1: Execution times and speedups for `./life3d-omp 3 1024 .4 100` with 1, 2, 4, 6 and 12 threads

From Figure 3, effective load balancing is observed as the number of simultaneously utilized CPUs matches the number of threads, except at the beginning where only one CPU is utilized. The time spent utilizing only one CPU remains approximately constant (around 15 seconds) and is attributed to the sequential portions of the code, such as generating the 3D grid. On the other hand, from Table 1, significant speedups (over 99%) are noted up to 6 threads. This aligns with anticipated results, considering the machine in use has 6 physical cores.

4 - Conclusions

This report aimed to optimize the 3D Game of Life simulation by applying parallel computing strategies, focusing primarily on OpenMP for enhancing performance. The main approach involved data decomposition and load balancing, dividing the computational grid into equal segments for processing by multiple threads. Synchronization techniques were carefully implemented to avoid data inconsistencies and race conditions, ensuring the integrity of the simulation's results. The static scheduling method was employed to distribute work among threads effectively, minimizing overhead and maximizing resource utilization. Empirical results, as demonstrated through execution times and speedups, indicate substantial performance improvements, particularly up to the number of physical cores available (6 in this case). Overall, the objectives were successfully met, showcasing the efficacy of parallel processing techniques in improving computational tasks.